11283/46

OPERATING SYSTEM HAVING A SYSTEM PAGE AND METHOD FOR USING SAME

INVENTOR(S):

MAARTEN A. KONING

PREPARED BY

KENYON & KENYON

333 W. SAN CARLOS ST., SUITE 600
SAN JOSE, CALIFORNIA 95110
408-975-7500

OPERATING SYSTEM HAVING A SYSTEM PAGE AND METHOD FOR USING SAME

BACKGROUND INFORMATION

Traditional multitasking operating systems (e.g., UNIX, Windows) have been implemented in computing environments to provide a way to allocate the resources of the computing environment (e.g., CPU, memory, Input/Output (I/O) devices) among various user programs that may be running simultaneously in the computing environment. The operating system itself comprises a number of functions (executable code) and data structures that may be used to implement the resource allocation services of the operating system.

Operating systems have also been implemented in a so-called "object oriented" manner. That is, when a particular function and/or data structure (defined by a "class" definition) is requested, the operating system creates ("instantiates") an "object" that uses executable code and/or data structure definitions specified in the class definition. Such objects thus may contain executable code, data structures, or both. Objects that perform actions are typically referred to as "tasks" (also known as "threads"), and a collection of tasks may be referred to as a "process." Upon loading and execution of the operating system into the computing environment, system tasks and data structures will be created in order to support the resource allocation needs of the system. User applications likewise upon execution may cause the creation of tasks ("user tasks"), processes ("user processes"), and other objects in order to perform the actions desired from the application.

In order to protect the operating system and each task running in the computing environment from interference from other tasks also running in the computing environment, typical operating systems apportion the computing environment's execution "space" (e.g., its memory) into a "system" space and a "user" space. The system space generally contains the operating system tasks and data structures, while the user space

contains the code and data structures for user tasks. Typically, operating systems are designed so that user tasks cannot directly access the memory apportioned to system tasks. The operating system itself, however, can typically access all portions of memory.

Conceptually, this "protection model" is illustrated by Figure 1. In a computer system 1 controlled by an operating system 2, there may be any number of user processes 3 and system tasks 5 executing at one time. User processes 3 each include a number of user tasks 6. Because each user process 3 is only allocated a portion of the system memory, the operating system 2 may restrict access by any user task 6 affiliated with a particular user process 3 to the memory allocated to another user process or the operating system. Typically, however, system tasks 5 have unrestricted access to memory allocated to the operating system 2 and each user process 3 (indicated by direct connections 7).

There may be instances, however, when a user task desires access to resources controlled by the operating system. For example, a user task may want access to a network I/O connection, the control of which is delegated to a system task. Alternatively, a user task often may want to read information from a data structure maintained and used by the operating system (e.g., an error status report, or the time remaining until some pre-planned system event occurs). Using known operating systems, in order to make such access, the user task is required to request execution of the system functions that perform the desired actions or control access to the desired data structures via a "system call". A system call is typically implemented via a special instruction used to cause the processor executing the code to "trap" to a trap routine (implemented in the system software) that makes a function call to the desired facilities of the operating system. Thus, the user task executing the system call cannot directly access the instructions and data structures in the system space it wishes to access, but rather must employ a special access procedure (represented in Figure 1 as connections 4). Furthermore, since the user task is not permitted access to system resources or data structures, another task must be created by the operating system in the system space in order to perform the requested action. While this procedure protects the operating system from potential interference caused by user tasks, it increases system-processing overhead and thus increases execution time.

Certain operating systems, called "real-time operating systems," have been developed to provide a more controlled environment for the execution of application programs. Real-time operating systems are designed to be "deterministic" in their behavior, i.e., responses to an event can be expected to occur within a known time of the occurrence of the event. Determinism is particularly necessary in "mission-critical" applications, although it is generally desirable for all operating systems, as it increases the reliability of the system. Real-time operating systems are therefore implemented to execute as efficiently as possible with a minimum of overhead.

Particularly in real-time systems, it is common and desirable for user tasks to obtain information about the overall state of the operating system, or information about particular aspects of the operating systems operations. This information allows tasks to operate more efficiently and reliably. For example, in a real time application, where a user task must return an answer by a deadline, a user process may choose a different method of computing a result depending on how much CPU time the user process expects to receive. A faster less, accurate method of computing might be used when the system is busy and less computation time is available, while a more accurate, but resource-intensive method might be used when more system resources are available. Similarly, a user process might decide to operate in a "safe" mode when errors have occurred. Thus, it would be beneficial to implement a system and method whereby user tasks are able to access operating system information without incurring operation overhead (such as created using the system trap) or compromising system security.

SUMMARY OF THE INVENTION

According to the present invention, an exemplary computer system is described, comprising a memory space having a number of memory locations, an operating system, a software module located, a plurality of operating system data structures, a system page

including a subset of the plurality of operating system data structures, and a function located within the software module.

A first method is also described as part of the exemplary embodiment according the present invention. The method includes the steps of creating a task assigned to execute at least one function, assigning a memory access map to the task, including in the memory access map indications of read-only access to memory locations of a system page including operating system data structures, and allowing a read memory access by the task to the system page.

A second method is also described as part of the exemplary embodiment according to the present invention. The second method includes the steps of retrieving a software module having a symbol reference used by an instruction, resolving the symbol reference to obtain a symbol value for the symbol, and inserting a symbol value in the into the instruction.

BRIEF DESCRIPTION OF THE DRAWINGS

Figure 1 shows a block diagram of a prior art protection model.

Figure 2 shows a block diagram of an exemplary computer system according to the present invention

Figure 3 shows an exemplary memory space of the exemplary computer system of Figure 2, according to the present invention.

Figure 4 shows an exemplary system space in the exemplary computer system of Figure 2, according to the present invention.

Figure 5 shows a diagram of a system page in the exemplary computer system of Figure 2, according to the present invention.

Figure 6 shows an exemplary symbol table for the exemplary system page, according to the present invention.

Figure 7 shows a flowchart of an exemplary linking process according to the present invention.

Figure 8 shows a flowchart of an exemplary memory reference process, according to the present invention.

DETAILED DESCRIPTION

According to the exemplary embodiment of the present invention, an operating system is provided that includes a "system page." In the exemplary embodiment, the system page is part of the memory space allocated to the operating system. The system page contains a subset of operating system data structures ("variables") that may be used by the operating system to store state information. These variables can be both read and written by operating system tasks. In addition, according to the present invention, user tasks are allowed direct, read-only access to system page variables, without the need to access these system page variables through a system call. User tasks may therefore obtain system information stored in system page variables without incurring the execution overhead produced by using a system trap (thus increasing execution speed) and without allowing alteration of these variables (thus maintaining system security).

An exemplary embodiment of a computing environment including an operating system having a system page according to the present invention is illustrated by Figures 2-8. Figure 2 is a block diagram of the exemplary computing environment 100. Computing environment 100 includes a CPU 101, which is coupled to a physical memory system 102 and a number of I/O systems 103. Connection of the CPU 101 to the physical memory system 102 and the number of I/O systems 103 may be according to any of the well

known system architectures (e.g., PCI bus) and may include additional systems in order to achieve connectivity. I/O systems 103 may comprise any of the well-known input or output systems used in electronic devices (e.g., serial port, modem, network connection, mouse, display). Physical memory system 102 may include RAM or other memory storage systems for operational memory, and read only memory and/or other non-volatile storage systems for storage of software (an operating system, other applications) to be executed in computing environment 100. Alternately, software may be stored externally of computing environment 100 and accessed from one of the I/O systems 103 (e.g., via a network connection). CPU 101 may also include a memory management unit (MMU, not shown) for implementing virtual memory mapping, caching, privilege checking and other memory management functions, as is well known. In systems without virtual memory, a simpler form of memory management unit, sometimes called a guarded memory unit, may be used for implementing privilege checking.

Figure 3 illustrates an exemplary memory space 110 of exemplary computing environment 100. Memory space 110 is, for example, an addressable virtual memory space available in the computer system 100 (which may be equal to or greater than the amount of physical memory provided in computing environment 100, depending on system memory management implementations). Memory space 110 may also include memory locations assigned as "memory mapped I/O" locations, allowing I/O operations through the memory space 110.

Memory space 110 includes a system space 112 and a user space 111. The system space 112 is used by an operating system 113 that controls access to all system resources (such as physical memory system 102 and I/O systems 103). The operating system 113 includes functions (executable code) and data structures, as well as a number executing system tasks and system objects that perform system control functions (e.g., context switching between tasks). As shown in Figure 3, user space 111 may include a number of user software modules 114 that have been loaded into the memory space 110 after the operating system 113 has begun executing in system space 112. The size of the system space 112 and user space 111 may be dynamically altered by the operating system 113

according to, for example, the number of software modules present in the system (and the amount of executable code / data structures within the software modules).

Figure 4 shows a more detailed view of the system space 112 and operating system 113. The system space 112 includes operating system functions 121 which are, for example, executable code used by system tasks. The system space 112 also includes an operating system workspace 123, which contains various data structures used by system tasks in performing operating system functions – stacks, heaps, linked lists, pointers and other data structures. System tasks are permitted complete access to data structures and other system objects in system space 112.

Exemplary system space 112 also includes a "system page" 122. System page 122 includes a subset of the overall collection of data structures (variables) used by the operating system 113 to store system state information, and, in this exemplary embodiment, includes operating system data structures that may be frequently accessed by user tasks executing in the computing environment 100.

Figure 5 shows a more detailed view of the exemplary system page 122. As mentioned above, the system page 122 is used to store system information that may be requested by user tasks. In the exemplary embodiment, the system page 122 includes the following values: a currently executing task identifier 141, an interrupt counter 142 indicating the current nesting level of interrupts (i.e., how many interrupts deep is the current process on the interrupt stack), a 64-bit absolute time counter 143 and a 32-bit absolute time counter 144, which indicate the time since the system was turned on, and a pointer 145 to a per-thread storage area that a currently executing task can read and write. Other system data structures might advantageously be included in the system page 122, depending on the specific design of the operating system or the nature of the application. Although these variables are shown as a table in this exemplary embodiment as simple 4-byte and 8-byte data structures, the system page could be implemented with more complex data structures, as may be desired for particular system implementations.

System page 122 may be created, for example, during loading of operating system 113 into memory space 110. As part of the loading process, an operating system symbol table may be created to allow linking to operating system data structures and functions. Symbols for system page variables will be included in the operating system symbol table to allow linking by user functions (described below). An exemplary operating system symbol table 150 is shown in Figure 6. Symbols for system page data structures have entries 152 that include a field indicating that the symbol is a system page variable. As discussed below, a linker can examine this field to determine if user functions can link to the operating system symbol.

In a system with virtual memory, it is also advantageous to implement the exemplary system page 122 so that the system page is "page aligned," i.e., so that the system page data structures start at the beginning of a virtual memory page (as shown in Figure 5). Such alignment may allow the MMU of the computing environment 100 to be used to enforce access protection rules for the system page 122 at run time, as will be described below. Page alignment may be accomplished by many known methods, depending on the method used to implement virtual memory in the computer system that includes the system page. For example, if the C or C++ language is used, page alignment can be accomplished by defining the system page variables in a separate file that is simply located at the beginning of the data section and modifying the linker to force the data section to be page aligned. For performance reasons, it may also be advantageous to insure that the system page fits within a single virtual memory page (typically four kilobytes).

In the exemplary embodiment according to the present invention, user tasks executing in computing environment 100 may only access functions and data structures in system space 112 through the use of a "protected" access (such as a system trap). However, as part of this exemplary embodiment, user tasks are allowed read-only access to the memory locations associated with the system page 122. Specifically, functions in user software modules 114 may link to memory locations associated with the system page 122 at link time, and user tasks executing the functions of software modules 114 may be

allowed to perform memory reads to memory locations associated with the system page 122 during execution. Since user tasks are permitted to directly access the variables of the system page 122, no protected accesses are needed that could increase execution overhead. Furthermore, since user tasks are not permitted to perform memory writes to system page memory locations, system security is maintained.

Figure 7 illustrates a flow chart of an exemplary loading and linking process for a software module 114 of a user application in conjunction with the exemplary system page according to the present invention. This linking process may be performed by a linker/loader application, for example, provided with the operating system 113. In step 170, the linker/loader is invoked to load the software module into computing environment 100 and link the software module 114 to other software modules 114 already loaded into computing environment 100 and/or with the operating system 113. The software module may initially be stored within the physical memory system 102 of the computing environment 100 or outside the computing environment 100 (accessed through the I/O systems 103).

In step 172, the software module 114 is loaded into user space 111. Operating system 113 allocates a portion of the memory space 110 to the software module 114, and the instructions and data structures of the software module 114 are loaded into memory locations corresponding to the portion of memory space 110 allocated to the software module 114. Memory references in the software module 114 may be relocated based on the location of the software module 114 in the memory space 110, as is well known. In step 174, symbol references in the software module 114 are resolved by searching any symbol tables in the computing environment 100 for symbol definitions corresponding to the symbol reference. Symbol definitions found in symbol tables other than the operating system symbol table (step 175) are resolved (step 177). If these other symbol tables in the computing environment 100 do not include an entry for the symbol referenced, the operating system symbol table will be searched (step 176). If the symbol reference matches a symbol in the operating system symbol table (step 178), the symbol entry will be checked to see if an indication is present that the symbol is associated with a system

page variable (step 180). If so, the linker will resolve the symbol reference using the symbol value for the symbol (step 182). If not, the linker will not resolve the symbol reference (step 184), and may also indicate that a link could not be established (for example, by aborting the linking process or providing a message to the user). If the symbol reference is not found in the operating system symbol table, the linker will not resolve the symbol reference (step 184), and also may indicate that the link could not be established. In step 186, the software module is linked, for example, by inserting the symbol values into the appropriate instructions in the loaded software module.

The method of controlling memory referencing in the exemplary embodiment during task execution is illustrated in Figure 8. Initially, a user task is created (instantiated) by the operating system 113, for example, by request from a user to execute the software modules 114 loaded into the memory space 110 (step 200). The user task includes task control information, which may include a memory allocation map indicating the portion of the memory space 110 that may be accessed by the user task. One example of such a memory map is a bit map corresponding to each virtual page in the memory space 110, such that virtual pages accessible to the user task are indicated in the memory map as set bits. Another example is a set of memory addresses or ranges of memory addresses for the portion of the memory space 110 accessible by the user task. Alternately, the task control information for each task may not include a memory map, but each virtual page in memory space 110 may include indicators for user task accessibility. According to the exemplary embodiment, the memory map of user task includes those memory locations in the portion of the memory space 110 that have been allocated to the software modules 114 to be executed by the user task.

The operating system 113 also inserts into the task memory map the system page memory locations, thus allowing the task to directly access to the system page memory locations (i.e., without the need for a system call). However, the operating system sets indicators that only permit the task to obtain read-only access to the system page memory locations. In the current example, access indicators (e.g., flag bits) are associated with the system page memory locations indicating whether user tasks are permitted read-only access.

These indicators may be set prior to task creation (e.g., during memory allocation for the system page). Access indicators may also be included in the memory map of the task control block (allowing task-specific access to the system page), in which case these indicators may be set after task creation and inserted into the memory map.

In step 202, user task executes an instruction in a software module 114 that attempts to reference a memory location. This memory reference may be through the use of a direct reference to the memory location desired, by de-referencing a pointer, or other well-known software construct. In step 204, the memory reference is checked to see if the referenced memory location is in the portion of the memory space 110 accessible to the user task (as reflected by the memory map for the user task in this example). This memory checking may be accomplished, for example, using the facilities of the MMU of the computing environment 100, such as a standard translation look-aside buffer and memory page-mapping system. If the memory location is not within the portion of memory space accessible to the user task 114, a memory fault occurs (step 206), which may be handled by an exception handling routine within the operating system 113 (which, for example, may result in the termination of the task).

If the memory location is within the portion of the memory space 110 mapped to the user task, an additional check is made to determine if the memory access is to the system page 122 (step 208), for example, by checking the access indicators to determine if the memory location is marked as read-only for the task. If the memory access is not to the system page 122, the memory access is executed (step 210). If the memory access is to the system page 122, a further check is made to determine if the memory access by the user task is a read access or a write access (step 212). If the user task is attempting to read from the system page 122, such access is permitted and the memory access is executed (step 210). If the user task is attempting to write to the system page, a memory fault occurs (step 214) causing the execution of the appropriate exception handling routine by the operating system 113.

As can be seen from the description above, enforcement of system page protection may be performed using a minimum of overhead (particularly where an MMU is available in the computing environment) and regardless of the method in which the system page is referenced (direct, de-referenced pointer, register pointer, etc.). Thus the system page according to the present invention allows quick access by user tasks to selected operating system variables without compromising system security and without the need for protected accesses or additional tasks.

In the preceding specification, the invention has been described with reference to specific exemplary embodiments thereof. It will, however, be evident that various modifications and changes may be made thereunto without departing from the broader spirit and scope of the invention as set forth in the claims that follow. The specification and drawings are accordingly to be regarded in an illustrative rather than restrictive sense.